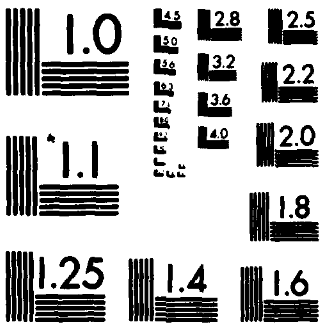
AD-A117 660 VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG --ETC F/G 9/2
THE DMS MULTIPROCESS EXECUTION ENVIRONMENT.(U)
APR 82 R W EHRICH N00014-81-K-0143
UNCLASSIFIED CSIE-82-6 NL

1 of 1

END
DATE
FILMED
8-8
DTI



MICROCOPY RESOLUTION TEST CHART

(1)

Virginia Polytechnic Institute and State University

Computer Science
Industrial Engineering and Operations Research
BLACKSBURG, VIRGINIA 24061

CSIE-82-6 ✓ April 1982 (1)

THE DMS MULTIPROCESS EXECUTION ENVIRONMENT

Roger W. Ehrich

TECHNICAL REPORT

Prepared for
Engineering Psychology Programs, Office of Naval Research
ONR Contract Number N00014-81-K-0143
Work Unit Number NR SRO-101

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CSIE-82-6	2. GOVT ACCESSION NO. AD-A117660	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE DMS MULTIPROCESS EXECUTION ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Roger W. Ehrich		8. CONTRACT OR GRANT NUMBER(s) N00014-81-K-0143
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Virginia Polytechnic Institute & State University Blacksburg, VA 24061		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N42 RR0420901 NR SRO-101
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 442 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE April 1982
		13. NUMBER OF PAGES 38
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

DMS, dialogue management system, multiprocessing concurrency, human-computer, dialogue independence, rendezvous, synchronization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this report a language-independent multiprocessing environment for managing human-computer dialogue is described. One of the main reasons for such an environment is that it helps to enforce the separation of modules that deal with human-computer dialogue from those that deal with computation while at the same time providing for concurrencies that would not otherwise be possible. Embedded in the environment are debugging aids that facilitate software preparation and a priority mechanism for sequencing interprocess communications.

ACKNOWLEDGEMENTS

This research was supported by the Office of Naval Research and ONR Contract Number N00014-81-0143, and Work Unit Number NR SRO-101. The effort was supported by the Engineering Psychology Programs, Office of Naval Research, under the technical direction of Dr. John J. O'Hare.

TABLE OF CONTENTS

Acknowledgements....................................ii

1. Introduction.....................................1

2. Functional Decomposition of Programs.............2

3. Interprocess Communication.......................9

4. Concurrency......................................13

5. Input and Output.................................15

6. The DMS User Procedures..........................17

6.1 Accept..17
6.2 Collect...17
6.3 Create_Process..................................17
6.4 Delete_Process..................................18
6.5 End_Request.....................................18
6.6 Input...18
6.7 Meter...19
6.8 Open_Meter......................................20
6.9 Output..20
6.10 Receive..20
6.11 Request..21
6.12 Send...21
6.13 Status_Report..................................22
6.14 Terminate......................................22
6.15 Trace..23

7. Using DMS..24

7.1 A FORTRAN Example...............................24
7.2 A VAX PASCAL Example............................27
7.3 Another FORTRAN Example.........................29
7.4 Debugging Concurrent Programs...................30
7.5 Running DMS on the VAX..........................30
7.6 Interacting with DMS............................32

References..34

1. INTRODUCTION

A Dialogue Management System (DMS) is a special system for creating, modifying, and testing human-computer interfaces. In general, DMS consists of three aspects -- an execution environment for dialogue and application software, a set of tools for creating software and human-computer interfaces, and a methodology for creating software using the tools and the execution environment. This report is a technical description of the DMS execution environment, and it includes the details of the FORTRAN procedures by means of which software is created to run under the DMS system. These procedures, in addition to the constructs of a standard programming language such as PASCAL or FORTRAN, are used to implement human-computer interfaces at an intermediate implementation level. Later, the DMS tools will provide high-level techniques for producing such interfaces without requiring the authors of those interfaces to be aware of the details at the intermediate level.

The system described here is based upon the belief of the DMS group at Virginia Tech that there should be logical and physical separation between software that implements the dialogues between human and computer and the software that achieves the computational goals of a computing system. The former are referred to as dialogue programs, and the latter are called computational programs.

2. FUNCTIONAL DECOMPOSITION OF PROGRAMS

DMS was motivated by a desire to improve the design and management of human-computer interfaces by providing a special environment for software production activities. Hartson, Ehrich, and Roach [1] point out that one of the fundamental reasons for the existence of so many poorly designed and inflexible human-computer interfaces is that current software design methodology encourages the production of program megaliths in which the software components are strongly coupled and interwoven. In particular, the failure to differentiate between software that implements the human-computer interface and software that implements the computational tasks has led to many of the problems. One of those problems is the failure to recognize that the design team having the competence to produce computational software frequently does not have the same competence in human factors and communication. A corollary is that many design teams do not fully grasp the difference. A second problem is that the software design tools that exist for application software design are not likely to be the same tools that are needed for the design of the human-computer interface. Yet a third consequence of monolithic software design is an inflexibility that may require considerable redesign when relatively small details of either application or interface software need to be changed.

Depending upon the nature of the software task, there may be a very close relationship between the design of the human-computer interface and the design of the computational software. Thus, the dialogue author and the computational programmer may need to work jointly on significant portions of the design specifications. We can clearly distinguish two extreme types of software, called computation dominant and dialogue dominant software. In the case of computation dominance, the control logic is entirely contained in the logic of the computational component so that whatever dialogue logic exists is distinct and easily decomposed from the logic of the computational component. Dialogue dominance is just the opposite. In this case the control logic is entirely contained in the dialogue, and again the computation and dialogue components are easily decomposed. Most text editors would resemble the dialogue dominant case. Most other software is somewhere in between, and its design requires the cooperation of the dialogue author and the programmer of the computational component. What is important is not the classification of a software task but the recognition of the distinct roles of the dialogue author and the programmer in the software production process. If one is willing to acknowledge these distinct roles, then it does not require much further argument to reach the conclusion that the objects designed by these authors ought also to be locally, logically distinct despite the global interrelationships that bind them together. It is the function of the DMS execution environment to provide the mechanism by means of which this logical separation can be implemented.

There are several ways to achieve separation of computational modules from dialogue modules. For example, a program might simply make calls to procedures that contain dialogue or computational code but not both together. Another alternative is to structure programs so that dialogue and computational components are distinct despite their presence in the same code body. Yet another solution makes use of multiprocessing to enforce separation and to make possible dialogue and computational concurrencies that would not otherwise be possible. This last alternative is the one that has been selected.

One of the most powerful concepts for implementing locally independent but globally interrelated software units is the process. A process provides the *environment*, services, and resources necessary for running a program, which, in the Digital Equipment Corporation (DEC) tradition, will be called an image to distinguish the bound executable module from source or object code. A process exists solely for the purpose of executing an image, which is both a logically and physically distinct software entity, much like that which a dialogue author or programmer would produce. The physical realization of the global links between dialogue or computational software modules is the interprocess communication facility, and it is partly the role of DMS to provide well engineered interprocess communication constructs within the context of the host operating system.

The principal consequence of DMS methodology is that a task that might be implemented as a single program under conventional software methodology will normally be implemented as a set of independent communicating programs, each of which executes in a separate process. Such a set of communicating programs will be referred to as a program complex. There are numerous benefits from this type of program decomposition. Module interactions are minimized, and that facilitates the design of the programs that run under the various processes. Concurrencies among modules become a reality, and since dialogue and computational code may well require different implementation tools, more specialized tools can be applied. The way in which a task is decomposed depends upon the control structures within the algorithm that carries out the task. For example, dialogue tasks are separated from computation tasks, and the interrelationships between them depends upon whether the overall task is dialogue or computation dominant. Thus, modifications to a software unit may be made by modifying any of the programs of the complex individually or by altering the communication among the programs.

In order to account for the different possible relationships among the programs of a complex it is necessary to distinguish special types of programs called executors and coprograms. These are distinguished on the basis of their internal structure, and they may implement either dialogue or computation, depending upon whether or not they are permitted to communicate with a user. Thus there may be dialogue programs, executors, and coprograms, and computation programs, executors, and coprograms. Executors have a particularly important structure; they are collections of modules that do not call one another and have no shared global environment. The module interrelationships are defined by the programs that invoke them. Few programs are organized as pure executors, but they contain an asymptotic structure that is important because of the independence of the component modules.

Programs and executors are used in situations where there is clear computation or dialogue dominance. The dominant task is implemented as a program that makes specific requests of the executor, which achieves logically distinct subtasks and returns information to the requesting program. While specific examples are shown later, the idea is that the executor is subordinate to the requesting program, and it exists to satisfy the program's

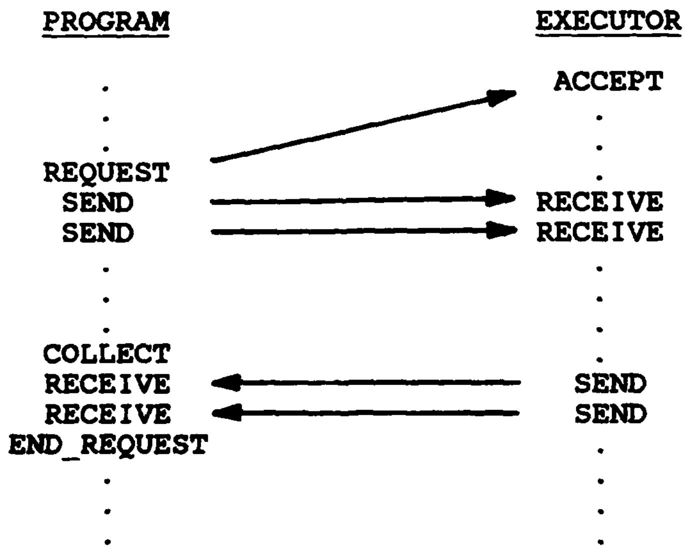


Figure 1 - Program-Executor Relationship

requests. The protocol that has been implemented looks roughly as shown in Figure 1. REQUESTs and ACCEPTs, and RECEIVEs and SENDs are complementary. The program makes a request of an executor which is acknowledged when the executor reaches its accept state. Then the program sends data to the executor and, when it desires the executor's response, collects the responses by issuing receives. The executor selects the appropriate service (e.g. dialogue module) specified by the request, issues matching receives and sends to obtain and return information, and performs any necessary input, output, or computation. Either side of the diagram in Figure 1 might contain computational or

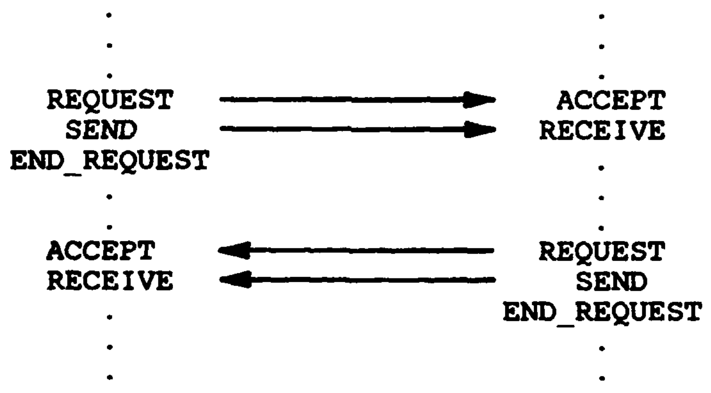


Figure 2 - Coprogram Relationship

dialogue code.

One possible organization of the coprogram relationship is illustrated in Figure 2. This relationship applies when there is no clear computation or dialogue dominance. In this case information is simply shuttled back and forth between the coprograms which execute in a quasi-synchronous manner. Of course, any program, coprogram, or executor may issue requests to any number of programs. Each program is identified in the

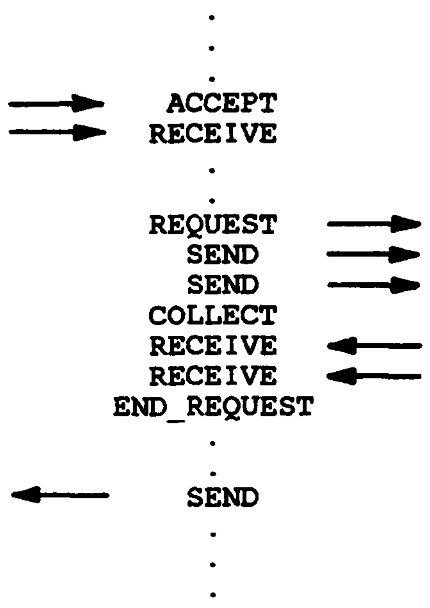


Figure 3 - Third-Party Request

request by the image name.

One other structure called a third-party request is possible. This situation, shown in Figure 3, occurs when an executor inserts a request to another executor in the middle of its service to a program. Such a request may be made only after the executor completes its RECEIVE sequence and before beginning its SEND sequence. It is the author's responsibility to avoid circular 3rd party requests that may lead to a deadlock. In the same spirit it is the authors' responsibility to agree upon request names, to ensure that RECEIVEs and SENDs are properly paired, and to ensure that the correct amount of data is transferred.

3. INTERPROCESS COMMUNICATION

DMS has been implemented using a variation of the rendezvous concept [2] in which two processes must synchronize before an interprocess dialogue can be initiated. While there are some requirements that are better served by asynchronous communication, it was felt that synchronous constructs would be easier to implement and much less confusing to nonspecialists in concurrent programming. In this implementation, an image called ENTRY is run under the user's login process. ENTRY is the central executive of DMS, and all DMS processes are subprocesses of the process running ENTRY. Included in ENTRY's responibilities are:

1) Conducting the default dialogue for initiating the execution of a complex.

2) Maintaining the image name, process identification code, mailbox name, and status for each DMS process.

3) Supervising process and mailbox creation.

4) Supervising mailbox deallocation and process shutdown for both normal and abnormal termination.

5) Servicing queries about processes that permit images to locate one another and establish communications.

Since this is a VAX/VMS implementation, the interprocess communication mechanism is called a mailbox, which is a portion of non-paged physical memory. Mailboxes are treated as ordinary input/output devices, and requests are queued if they cannot be serviced immediately. Each process has one request mailbox whose name is based upon its process identification code, which is a unique number assigned by the operating system to the process when it is created. An image never writes to its own mailbox, which is treated as read only by its process. In each process except the one running ENTRY, mailboxes are serviced by software interrupts called asynchronous system traps (AST's). Thus mailboxes are read almost instantaneously with one exception. When data (as opposed to control information) are being transferred, since each image has only limited local buffer space, AST's are disabled in the receiving process until the received data can be copied to its final destination to free the receive buffer.

Two processes are required to synchronize at the beginning of each interprocess dialogue. If process A requests a dialogue with process B, A sends B a request and hibernates until B gives A its attention. Then A and B can communicate freely, but with two constraints. In order to receive information from B, A must issue a collection request to B at some point before information is desired from B. This permits A to initiate concurrent dialogues with several processes and then order the processes from which it desires responses. The second constraint is that if B makes a third party request to C, it may only do so when A has finished transmitting data to B. Otherwise B would not be able to distinguish data received from A from data received from C. With the exception of the preceding caution, A and B may freely exchange data in either direction as long as they choose.

As an executor receives requests, they are queued internally and then serviced according to their priority. Once a request is serviced, the rendezvous is in effect until both processes decide to end the dialogue. With such an implementation the organization of an executor is quite simple, since it queues only requests and carries out only one interprocess dialogue at a time.

To complete the description of the DMS execution environment, a brief account of the communications that take place internally is given next. Suppose that image A is a computational program and that image B is a dialogue executor. When a user runs ENTRY, the user gives A's name in the default dialogue, and its process is created. When A executes a REQUEST, A creates a mailbox and informs ENTRY that it has done so. Since A has never communicated with B before, A asks ENTRY for the name of B's mailbox. ENTRY, however, has no knowledge of B either, so it creates a process that runs B. As soon as B executes an ACCEPT, B creates a mailbox and informs ENTRY. Next, ENTRY checks whether any image needed B's mailbox name, and it immediately sends the information to A, whose process is hibernating. A wakes up and now requests service from B and waits for acknowledgement, which comes immediately, since B's process is hibernating. Then data exchanges follow. If A later requests service from B again, ENTRY is not queried, since A remembers B's mailbox name.

There are two ways in which processes are removed. If the DELETE_PROCESS service is called, ENTRY broadcasts a notice to all processes to cancel records of the deleted process, and then it deletes the process. If any image terminates normally or by accident, ENTRY learns of the event by receiving a message in a special mailbox called a termination mailbox. When such a message is received, ENTRY deletes all processes, sends any error message to the standard error device, and returns to the default dialogue so that the user can execute another complex.

4. CONCURRENCY

Up to now the merits of the multiprocess environment for DMS have been argued on the basis of the functional decomposition of programming tasks. However, there is far more involved. There is no difference in the theory if the individual processes were separate hardware processors. All of them may function concurrently with only weak constraints imposed in part to make the programmer's job easier. Most restrictive is the constraint that a process issuing a REQUEST will hibernate until that request is acknowledged. However, assuming that the required executors or co-programs are free, a program may initiate multiple dialogues that execute concurrently with one another and concurrently with its own code.

There exist several programming environments that provide multiple processes and interprocess communication facilities, among them ADA [2], concurrent PASCAL [3], and UNIX [4]. These differ from one another in their constructs and implementation, and since their definition is standard, one does not have the freedom to adapt them to the specific needs of dialogue management. One need that has been addressed, for example, is the notorious complexity of debugging asynchronous concurrent programs. Another need is satisfied by the provision of a built-in mechanism for executing queued requests on the basis of their priority. DMS is largely language independent and has an architecture that can easily be modified to meet the needs that arise as we learn how to provide better and better tools for constructing human-computer interfaces.

5. INPUT AND OUTPUT

The handling of I/O devices is one of the most complicated parts of implementing human-computer dialogues. In order to be able to provide intelligent input/output devices and software capable of interacting with many concurrent users or computational processes, each device is controlled by a separate program running in a distinct DMS process and served by the same interprocess communication facility. For each physical device, the DMS system is supplied with a set of high-level subroutines by means of which input/output requests are made to that device. Users of DMS are not aware that separate processes are created to serve input/output requests.

Every device handler is accessed through two basic subroutines, INPUT and OUTPUT, in addition to special subroutines designed to serve special device features and special device software capabilities. INPUT and OUTPUT are used to transmit ASCII data to and from a device, and if these subroutines are used, formatting must be done by dialogue processes. As DMS evolves, INPUT and OUTPUT will be supplemented by higher-level subroutines that support intelligent formatting directly in the device process. Different physical device types are controlled by different device programs. However, if two different device types have similar capabilities, the same subroutines are designed to support them. This rather neatly solves the problem of device dependence in input and output.

One important aspect of each device process is the queueing of I/O requests from multiple processes. There is only a limited priority system for I/O requests. All requests are served on a first-come, first-served basis except for output requests with the FLASH mode specifier. Such requests are served first. In order to prevent undesired shifts of focus in a device dialogue, each communication with the device may use a KEEP mode specifier which blocks access to the device to all other processes. The KEEP is in effect only for the next communication unless the next communication also specifies KEEP. Even if FLASH is specified, if another process has reserved a device by specifying KEEP, the FLASH will be queued until the device becomes free.

6. THE DMS USER PROCEDURES

The fifteen procedures described here pass requests by character strings specified by descriptors and data by reference. Considerable effort has been made to simplify these services by eliminating arguments without sacrificing functionality and flexibility.

6.1 ACCEPT (request_name)

ACCEPT is the procedure that is executed in a dialogue or computation executor or co-program in order to obtain the name of a request for service from another program in the complex. Request_name is a character string up to 31 characters in length. Each call to ACCEPT in one process is logically paired with a corresponding REQUEST in the communicating process. The process containing ACCEPT waits until a service request has been received before returning. If multiple requests have been queued, the one with highest priority is acknowledged.

6.2 COLLECT (image_name)

COLLECT is called by a program requesting service of an executor to trigger the return of dialogue or computational results from the executor. Image_name is the name of the executor and can be up to 63 characters in length.

6.3 CREATE_PROCESS (image_name)

This subroutine can be called from any process to explicitly create a subprocess of the ENTRY process. Image_name is the name of the image to run under the new process, and its name must have an "exe" filetype. Image_name may be up to 63 characters in length. There is an implicit call to CREATE_PROCESS whenever a call to REQUEST is made specifying an image that is not currently running under some process.

6.4 DELETE_PROCESS (image_name)

This subroutine explicitly deletes the subprocess of the entry process that is running the image specified by image_name. Care must be taken not to delete a process that is engaged in an interprocess dialogue or an I/O request, since other processes participating in such a dialogue will not be able to proceed. Image_name may be up to 63 characters in length, and the image must have an "exe" filetype.

6.5 END_REQUEST

This should always be used as the last statement of a dialogue request. In other words, each call to REQUEST should eventually be followed by a call to END_REQUEST.

6.6 INPUT (device,string,nbytes,mode)

INPUT requests input data from 'device', which is specified by a 2-character string such as 'tt' for the user's login terminal. Nbytes is a 4-byte integer variable in the range 0 to 512 that specifies the buffer size, and it is set to the number of bytes returned, excluding line terminator, if any. Mode is a character string that determines how the input is to be done. A mode specifier can appear anywhere in the mode string, and it can have upper or lower case. However, mode cannot be a zero-length string. Valid mode specifiers are:

ECHO	-	Echo the input characters
KEEP	-	Retain device control after transmission
EDIT	-	Edit input as follows: CR and LF line terminators ¢U (NAK) cancel line ¢R (DC2) refresh input line DEL character delete
PURGE	-	Clear the typeahead buffer before input
CR	-	Carriage return after input
LF	-	Line feed after input
RESTORE	-	After input, erase line and restore cursor position
BELL	-	Ring the bell before accepting input

Example:

```
call INPUT ('tt',string,nbytes,'echo+keep')
```

6.7 METER (id,comment,p1,p2,p3,p4,p5)

A call to METER causes a record to be written to the metering file specified by the OPEN_METER subroutine. id and p1 - p5 are optional 2-byte integers, and comment is a character string of 1 to 35 characters. Each record has the following format:

2	cols	id	i2
11	cols	elapsed seconds	f11.2
2	cols	sp	
35	cols	comments	
5	cols	p1	i5
6	cols	p2	i6
6	cols	p3	i6
6	cols	p4	i6
6	cols	p5	i6
1	col	NULL	

6.8 OPEN_METER (file_name)

This subroutine prepares a file to accept 80-character metering records produced by calls to METER. The metering file must be created prior to a call to OPEN_METER. A record is simply appended to this file each time METER is called.

6.9 OUTPUT (device,string,nbytes,mode)

OUTPUT sends data to 'device', which is specified by a 2-character string such as 'tt' for the user's login terminal. Nbytes is a 4-byte integer in the range 0 to 512 that specifies the number of bytes to be transmitted. Mode is a character string that describes how the output operation is to be performed. A mode specifier can appear anywhere in the mode string, and it can have upper or lower case. However, it cannot be a zero-length string.

KEEP	-	Retain device control after transmission
CR	-	Carriage return after transmission
LF	-	Line feed after transmission
FLASH	-	Send a high priority message

Example:

```
Call OUTPUT ('tt',string,100,'crlf and flash')
```

6.10 RECEIVE (data_reference,nbytes)

RECEIVE implements the transfer of data from another process into the process from which it is called, and the amount of data transmitted is determined by the sending process. Data_reference is the address of a contiguous byte array that is to receive the transmitted data, and nbytes is a 4-byte integer that returns the number of bytes transmitted. Each call to RECEIVE in one process is logically paired with a corresponding SEND in the communicating process. RECEIVE waits until data have been received before returning.

6.11 REQUEST (image_name,request_name,priority)

REQUEST is called by a program that wishes to invoke dialogue or computation by an executor. Image_name is the name of the image from which services are requested. It must have an "exe" filetype and may be up to 63 characters in length. Request_name is a character string that names the request to be serviced by the executor, and it may be up to 31 characters in length. Each call to REQUEST in one process is logically paired with a corresponding ACCEPT in the communicating process. REQUEST will not return until the named image is running, its mailbox has been established, and it has acknowledged the request. If the named image is not running under any process, an implicit call to CREATE_PROCESS is made. Priority is a 4-byte integer that is used by ACCEPT to determine which request among those pending should be served next. Requests with larger priority values are served first.

6.12 SEND (data_reference,nbytes)

SEND transmits "nbytes" bytes of contiguous data from a data structure whose address is given by data_reference, where nbytes is a 4-byte integer. Each call to SEND in one process is logically paired with a RECEIVE in the communicating process.

SEND returns immediately if it is called after a REQUEST. When called after an ACCEPT, SEND returns immediately if the sequence of SENDs has been enabled by another process by a call to COLLECT. SEND should not be issued by a requesting process unless RECEIVE is the next subroutine to be executed by the accepting process. Otherwise the accepting process will be deadlocked.

6.13 STATUS_REPORT

Any process running under DMS can request that the status of all subprocesses with mailboxes be transmitted to the user's login terminal. The status report specifies which of the DMS services ACCEPT, COLLECT, END_REQUEST, RECEIVE, REQUEST, or SEND is currently being executed or was most recently executed. Where applicable, the name of the communicating image and the requested service is also returned. This subroutine is used for debugging purposes in determining the execution state of DMS subprocesses, and the same status report can be obtained from DMS interactively.

6.14 TERMINATE (mode)

Terminate calls for termination of the program complex currently running under DMS or causes the termination of DMS itself. Mode is a character string that determines the effect of TERMINATE - 'complex' if the current complex is to be terminated and 'dms' if DMS itself is to be terminated. Control of execution never returns from TERMINATE. Terminate can also be executed interactively through the DMS system.

6.15 TRACE (onoff)

Any process running under DMS can request an execution trace of the DMS services ACCEPT, COLLECT, END_REQUEST, RECEIVE, REQUEST, and SEND. When TRACE is called, the trace begins for all processes that have a mailbox at the time the request is issued. Onoff is a character string that turns trace mode on if 'ON' and off if 'OFF'. A trace is a time-sequential transcript that shows the entry and exit from each of the above six DMS subroutines and a record of each interprocess communication sent to a process mailbox. This subroutine is used for debugging purposes in determining the execution state of DMS subprocesses. Trace can also be turned on or off from DMS interactively, and all trace output is sent to the file, TRACE.DMS.

7. USING DMS

This section presents short examples of communicating programs written in FORTRAN and PASCAL that demonstrate how the DMS multiprocess environment is to be used. Also, this section contains specific notes about the VAX/VMS environment which hosts the DMS system.

7.1 A FORTRAN Example

```
c     This program gets a number and doubles it.
c     Let's assume that this program is called FDEMO.
c     According to DMS methodology, this program may not
c     communicate with a user.
c     This program executes in a subprocess of the user's
c     login process concurrently with FEXEC.

      call REQUEST ('fexec','get_number',0)
      call COLLECT ('fexec')
      call RECEIVE (n,nbytes)
      call END_REQUEST

      m=2*n

      call REQUEST ('fexec','print_numbers',0)
      call SEND (n,4)
      call SEND (m,4)
      call END_REQUEST

      call REQUEST ('fexec','shutdown',0)
      call END_REQUEST

      end
```

```
c     This is a dialogue executor that serves the requests
c     from FDEMO and does all its input and output.
c     Let's assume that this executor is called FEXEC.
c     This program executes in a subprocess of the user's
c     login process concurrently with FDEMO.

      CHARACTER*31 request_name

10    call ACCEPT (request_name)

      if (request_name.eq.'get_number') then
        type *, 'Enter an integer'
        read *, n
        call SEND (n,4)
      end if

      if (request_name.eq.'print_numbers') then
        call RECEIVE (i,nbytes)
        call RECEIVE (j,nbytes)
        type *, '2',i,' equals',j
      end if

      if (request_name.eq.'shutdown') call exit

      go to 10

      end
```

Looking first at FDEMO, the first request is for an input value to be used in a simple computation. Since FDEMO has nothing to send the dialogue executor, it calls COLLECT and obtains the value from FEXEC. Then, a request is made for FEXEC to print the input and output values. Since they are not necessarily contiguous in storage, two calls are made to SEND, each having 4 bytes since the default FORTRAN integer data type is INTEGER*4. Finally, the program complex is terminated by the REQUEST for the shutdown service. Since the requests are executed on a first-come, first-served basis, the shutdown request which causes all programs in the complex to terminate will not be executed by FEXEC until it has finished printing out the results for the user.

FEXEC begins with an ACCEPT statement, and once a request_name is obtained, the corresponding request is executed. The only unusual service is shutdown, which causes a program exit, which, in turn, causes the entry process to delete all processes in the complex and return to the default dialogue.

The following examples are identical to the previous ones except that both the application program and the dialogue executor have been coded in VAX PASCAL. Otherwise, the previous discussion applies also to these programs.

7.2 A VAX PASCAL Example

```
program pdemo;
{  This program gets a number and doubles it.
According to DMS methodology, this program may not
communicate with a user.
This program executes in a subprocess of the user's
login process concurrently with PEXEC.}

type
 executor_name_type = packed array [1..5] of char;
 request_name_type = packed array [1..13] of char;

var
 m,n,priority,length,nbytes: integer;

procedure REQUEST (%stdescr u: executor_name_type;
                  %stdescr v: request_name_type;
                  var w: integer); extern;
procedure COLLECT (%stdescr u: executor_name_type); extern;
procedure RECEIVE (var u: integer;
                  var v: integer); extern;
procedure END_REQUEST; extern;
procedure SEND (var u: integer; var v: integer); extern;

begin
 length:=4;
 priority:=0;

 REQUEST ('pexec','get_number   ',priority);
 COLLECT ('pexec');
 RECEIVE (n,nbytes);
 END_REQUEST;

 m:=2*n;

 REQUEST ('pexec','print_numbers',priority);
 SEND (n,length);
 SEND (m,length);
 END_REQUEST;

 REQUEST ('pexec','shutdown     ',priority);
 END_REQUEST

end.
```

```
program pexec (input,output);
{  This is a dialogue executor that serves the requests
from PDEMO and does all its input and output.
This program executes in a subprocess of the user's
login process concurrently with PDEMO.}

type
 request_name_type = packed_array [1..13] of char;

var
 request_name: request_name_type;
 i,j,n,length,nbytes: integer;

procedure ACCEPT (%stdescr u: request_name_type); extern;
procedure SEND (var u: integer,var v: integer); extern;
procedure RECEIVE (var u: integer;
                   var v: integer); extern;
procedure SYS$EXIT; extern;

begin
 length=4;

 repeat
   ACCEPT (request_name);
   case request_name [1] of

     'g':
       begin
         writeln ('Enter an integer');
         readln (n);
         SEND (n,length)
       end;

     'p':
       begin
         RECEIVE (i,nbytes);
         RECEIVE (j,nbytes);
         writeln ('2',i,' equals',j)
       end;

     's': SYS$EXIT

   end
 until false

end.
```

One more example is given that demonstrates the use of the DMS I/O system. In this example, a line of up to 25 characters is typed with echo to the user. When carriage return is struck, the line is repeated on the user's terminal. If the first character is an uppercase 'S', DMS is terminated.

7.3 Another FORTRAN Example

```
c     This program parrots the input typed on the user's
c     terminal.  If 25 or more characters are typed or if
c     CR is struck, the line typed is retyped.

      CHARACTER*25 buffer

10    nbytes=25
      call INPUT ('tt',buffer,nbytes,'EDIT + ECHO + CRLF')
      call OUTPUT ('tt',buffer,nbytes,'CRLF')

      if (buffer(1).eq.'S') call TERMINATE ('dms')

      go to 10

      end
```

7.4 Debugging Concurrent Programs

STATUS_REPORT and TRACE will provide static and dynamic information about interprocess communication in DMS. The TRACE result is a readable transcript that shows a time-sequential history of information transmission and subroutine execution. Sometimes the order of items in the transcript may seem peculiar. This is because in a timesharing system only one process can actually execute at a time while others await their turn. The TRACE result is sent to the file, TRACE.DMS, where it is saved for subsequent analysis.

7.5 Running DMS on the VAX

Before running DMS on VAX1, all the programs that are to run under its control need to be linked to the DMS library. The two most common ways of doing the link would be either to specify the DMS library explicitly in the link command or to define a link library in the LOGIN.COM file. The two alternatives are

```
(1)  LINK  PROGRAM,[EHRICH.DMS5]DMS/LIB
(2)  DEFINE  LNK$LIBRARY  [EHRICH.DMS5]DMS
```

The name of every program running under DMS is mapped through logical-name translation. If the DMS user wishes to run two different programs in a process without recoding the program references in the complex, logical names can be used. If no logical name exists, the given name is used. Suppose, for example, that a complex references a program called JUDY. If the user wishes to substitute GEORGE for JUDY, it is only necessary to type

```
ASSIGN  george  judy
```

before executing DMS. Of course, in this case the file named GEORGE could have been renamed JUDY, but that is not always possible if GEORGE is in another directory.

For each I/O device xx referenced in a program complex, two things are required for execution of that complex:

1) A logical name xx that translates to the physical I/O device (like _TTA4:)

2) A driver (or a logical name that translates to a driver) whose name is XXDRIVER.

The only exception is TT, which by default translates to the user's login terminal.

Logical names can also be used to run multiple devices with one driver program. Suppose three identical devices are required whose drivers are GGDRIVER, and suppose that in the program complex, the devices are referenced by G1, G2, and G3. The translations for G1DRIVER, G2DRIVER, and G3DRIVER are all set to be GGDRIVER, and the translations for G1, G2, and G3 are set as usual to be the physical terminal line names.

To run DMS, type

```
R  [EHRICH.DMS5]ENTRY
```

ENTRY is the name of the central program that supervises the execution of all the images in its subprocesses. When ENTRY runs, the prompt

```
          Welcome to DMS: enter the image name
```

will be typed, and all the user need do is specify the name of the image that starts the execution of the program complex. In the case of the FDEMO and FEXEC programs given previously, FDEMO is all that need be typed. Typing FEXEC will not work since FEXEC contains no reference to FDEMO that would cause it to begin execution. Remember also that if any program in the complex terminates in any way except by a call to DELETE_PROCESS, ENTRY will terminate the entire complex and return to the default dialogue.

7.6 Interacting with DMS

When a DMS user types ^C, ENTRY types the prompt DMS>, which is a request to the user to issue a command to the DMS system itself. While in DMS command mode, all processes are active, but DMS output (ie, output sent through a DMS device process) to the login terminal is blocked. When ^C is typed, current output-requests are terminated, and any pending read-request will be reissued. If any FORTRAN or PASCAL read-requests to the login terminal are active, an additional CR will be required to get the DMS> prompt. Any such read request will subsequently be reissued.

The DMS commands are:

Quit, STOp, Exit, TErminate DMs	to leave DMS
TErminate COmplex	to reinitiate DMS
TRace ON	to turn trace on
TRace OFf	to turn trace off
STAtus report	to get a status report
<CR>	to leave interactive mode

DMS requires certain privileges and quotas. These include

```
PRCLM  =  nproc
BYTLM  =  2500 * (nproc + 1)
TMPMBX
GRPNAM
GROUP
```

where nproc is the maximum number of concurrent programs in the complex to be run.

Users should use the hibernate system service (sys$hiber) with caution since any communications to the process mailbox such as a service request will cause a wakeup to be issued by the AST that services the mailbox. For applications requiring time delays, procedures such as DELAY (<100 seconds) or LONG_DELAY (>1 second) should be used since they use the sys$setimer service. These are also found in the DMS library.

REFERENCES

1. Hartson, H.R., Ehrich, R.W., and Roach, J. The Management of Dialogue for User/Software Interfaces, (Technical Report). In preparation.

2. United States Department of Defense. Reference Manual for the Ada Programming Language. DARPA, July 1980.

3. Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, 1975, SE-1 (2), 199-207.

4. Ritchie, D.M. and Thompson, K. The UNIX Timesharing System, Comm. ACM, 1974, 17 (7), 365-375.

OFFICE OF NAVAL RESEARCH

Code 442

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

Capt. Paul R. Chatelier
Office of the Deputy Under Secretary of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

Department of the Navy

Engineering Psychology Programs
Code 442
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Communication & Computer Technology Programs
Code 240
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Tactical Development & Evaluation Support Programs
Code 230
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Manpower, Personnel and Training Programs
Code 270
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Information Systems Program
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Physiology & Neuro Biology Programs
Code 441B
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

Special Assistant for Marine Corps Matters
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Commanding Officer
ONR Eastern/Central Regional Office
ATTN: Dr. J. Lester
495 Summer Street
Boston, MA 02210

Commanding Officer
ONR Western Regional Office
ATTN: Dr. E. Gloye
1030 East Green Street
Pasadena, CA 91106

Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco, CA 96503

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D.C. 20375

Dr. Louis Chmura
Code 7592
Naval Research Laboratory
Washington, D.C. 20375

Dr. Robert G. Smith
Office of the Chief of Naval Operations, OP987H
Personnel Logistics Plans
Washington, D.C. 20350

Department of the Navy

Dr. Jerry C. Lamb
Combat Control Systems
Naval Underwater Systems Center
Newport, RI 02840

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Human Factors Department
Code N-71
Naval Training Equipment Center
Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training Equipment Center
Code TAEG
Orlando, FL 32813

Dr. Albert Colella
Combat Control Systems
Naval Underwater Systems Center
Newport, RI 02840

K. L. Britton
Code 7503
Naval Research Laboratory
Washington, D.C. 20375

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Mr. Warren Lewis
Human Engineering Branch
Code 8231
Naval Ocean Systems Center
San Diego, CA 92152

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Department of the Navy

HQS, U.S. Marine Corps
ATTN: CCA40 (MAJOR Pennell)
Washington, D.C. 20380

Commanding Officer
MCTSSA
Marine Corps Base
Camp Pendleton, CA 92055

Chief, C^3 Division
Development Center
MCDEC
Quantico, VA 22134

Dr. Robert Wisher
Naval Material Command
NAVMAT 0722 - Rm 508
800 North Quincy Steet
Arlington, VA 22217

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 340F
Washington, D.C. 20361

Commander
Naval Air Systems Command
Crew Station Design
NAVAIR 5313
Washington, D.C. 20361

Mr. Phillip Andrews
Naval Sea Systems Command
NAVSEA 0341
Washington, D.C. 20362

Commander
Naval Electronics Systems Command
NC #1, Rm 4E56
Code 81323
Washington, D.C. 20360

Dr. Arthur Bachrach
Behavioral Sciences Department
Naval Medical Research Institute
Bethesda, MD 20014

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research Lab.
Naval Submarine Base
Groton, CT 06340

Department of the Navy

Head
Aerospace Psychology Department
Code L5
Naval Aerospace Medical Research Lab.
Pensacola, FL 32508

Dr. James McGrath
CINCLANT FLT HQS
Code 04E1
Norfolk, VA 23511

Navy Personnel Research and
Development Center
Planning & Appraisal Division
San Diego, CA 92152

Dr. Robert Blanchard
Navy Personnel Research and
Development Center
Command and Support Systems
San Diego, CA 92152

LCDR Stephen D. Harris
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dr. Julie Hopson
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Mr. Jeffrey Grossman
Human Factors Branch
Code 3152
Naval Weapons Center
China Lake, CA 93555

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93042

Mr. J. Williams
Department of Environmental
Sciences
U.S. Naval Academy
Annapolis, MD 21402

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

Department of the Navy

Human Factors Section
Systems Engineering Test
Directorate
U.S. Naval Air Test Center
Patuxent River, MD 20670

Dr. Robert Carroll
Office of the Chief of Naval
Operations (OP-115)
Washington, D.C. 20350

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, D.C. 20310

Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Director, Organizations and
Systems Research Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs.
Aberdeen Proving Ground, MD 21005

ARI Field Unit-USAREUR
ATTN: Library
C/O ODCSPER
HQ USAREUR & 7th Army
APO New York 09403

Department of the Air Force

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D.C. 20332

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Department of the Air Force

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks AFB, TX 78235

Foreign Addressees

North East London Polytechnic
The Charles Myers Library
Livingstone Road
Stratford
London E15 2LJ
ENGLAND

Professor Dr. Carl Graf Hoyos
Institute for Psychology
Technical University
8000 Munich
Arcisstr 21
FEDERAL REPUBLIC OF GERMANY

Dr. Kenneth Gardner
Applied Psychology Unit
Admiralty Marine Technology
Establishment
Teddington, Middlesex TW11 OLN
ENGLAND

Director, Human Factors Wing
Defence & Civil Institute of
Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

Dr. A. D. Baddeley
Director, Applied Psychology Unit
Medical Research Council
15 Chaucer Road
Cambridge, CB2 2EF
ENGLAND

Prof. Brian Shackel
Department of Human Sciences
Loughborough University
Loughborough, Leics, LE11 3TU
ENGLAND

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314

Dr. Craig Fields
Director, System Sciences Office
Defense Advanced Research Projects
Agency
1400 Wilson Blvd.
Arlington, VA 22209

Dr. M. Montemerlo
Human Factors & Simulation
Technology, RTE-6
NASA HQS
Washington, D.C. 20546

Other Organizations

Dr. Jesse Orlansky
Institute of Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

Dr. Robert T. Hennessy
NAS - National Research Council
Committee on Human Factors
2101 Constitution Ave., N.W.
Washington, D.C. 20418

Dr. Elizabeth Kruesi
General Electric Company
Information Systems Programs
1755 Jefferson Davis Highway
Arlington, VA 22202

Mr. Edward M. Connelly
Performance Measurement
Associates, Inc.
410 Pine Street, S.E.
Suite 300
Vienna, VA 22180

Dr. Richard W. Pew
Information Sciences Division
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238